Experimental research of optimizing the Apache Spark tuning: RDD vs Data Frames

Sergii Minukhin[1[0000-0002-9314-3750]], Maksym Novikov[1[0000-0002-8978-044X]], Natalia Brynza[1[0000-0002-0229-2874]], Dmytro Sitnikov[2[0000-0003-1240-7900]]

[1]Simon Kuznets Kharkiv National University of Economics, 9a Nauka Ave., Kharkiv 61166 Ukraine
minukhin.sv@gmail.com, maksym.novikov@hneu.net, natalia.brynza@hneu.net

[2]Harkiv National University of Radio Electronics, 14 Nauka Ave., Kharkiv 61166, Ukraine
dmytro.sytnikov@nure.ua

Abstract. In this paper results and analysis of experimental research for determining the effectiveness of changing the parameters (as compared to standard values) of tuning Apache Spark for minimizing application execution time have been presented. The structure of a test dataset has been developed using RDD and Data Frames, based on which it is possible to create during a minimal time text files with a size greater than 4 GB having properties (characteristics) set up for testing. A peculiarity of test data is the fact that they often reflect basic properties of real world problems. The investigation includes 2 stages: at the first stage a comparative analysis of RDD and Data Frames is carried out for the standard settings of Apache Spark; at the second stage experiments for different sizes of an input test dataset for assessing the influence of parallelism levels, a block size in HDFS and the parameter spark.sql.shuffle.partitions in Spark Data Frames have been conducted. The obtained results substantiate the influence of the spark.sql.shuffle.partitions value on the test task execution performance. For this parameter ranges and change trends have been found. Also, levels of parallelism that maximally influence the execution time have been determined. It has been proven that for certain sizes of input test files the size of an HDFS block can be set up by default. Results of computational experiments have been demonstrated in tables and graphs. They confirm the effectiveness of the suggested changes to the Apache Spark settings as compared with the standard ones for different sizes of tested files.

Keywords. Apache Spark, Resilient Distributed Dataset, Data Frames, HDFS, shuffling, level of parallelism, data processing, data set, application, execution time.

Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2608/paper31.pdf

1 Introduction and related works

Spark is a general-purpose, fault-tolerant, and fast cluster computing platform for processing big data. Apache Spark is a popular platform for data analytics that has been used by many organizations (Yahoo!, eBay, Baidu, Netflix [1]) owing to its in-memory computing framework. The Apache Spark framework is open source. It offers a number of tuning capabilities and can be customized. Its configuration has a 100 parameters that can be easily set up and changed by the user and tailored according to the cluster and application he uses [2]. The main types and description of Apache Spark parameters are presented in Table 1 [2, 3].

Table 1. The main types and description of Apache Spark parameters

Parameter type	Description
Application Properties	Related to basic properties of an application, for example, the application name, the CPU and the memory resources that are supposed to be allocated to the coordinating process (driver), the memory allocated for all executor processes that perform computations
Runtime Environment	Settings for environment (class paths, java options and logging)
The level of parallelism	It relates to the number of tasks in which each Resilient Distributed Dataset (RDD) is divided into parts. It should be set in a way that the resources of a cluster are completely utilized
Shuffle Behavior	Such parameters represent the shuffling mechanism of Spark, and they include methods of shuffling, buffer settings, sizes, memory selected for shuffling
Execution Behavior	Fractions of memory, number of execution cores and the level of parallelism
Scheduling	Related to the scheduling mode and define, in particular, the maximum number of CPU cores that will be used
Networking	Related to timeout options, ports, optimal values for network retries, heartbeat pauses
Compression and Serialization	These parameters define if compression is applied or not, which compression codec should be used, parameters of the codec, the serializer to be used and its buffer options
Spark UI	These parameters are mainly associated with UI event-logging

The existing methodological approaches to optimizing settings (resetting standard values) of the Spark parameters allow increasing computational performance for solving problems related to various fields are considered in [1-3]. However, owing to

substantial differences in applications and their different performance (with respect to input data characteristics), developing a generalized algorithm that works effectively for any application is difficult [1]. Thus it is important to improve the execution performance of Spark for different application types, which can be done with the help of: optimizing the physical execution plan of a Spark job; efficiently scheduling parts of a Spark job on cluster nodes; and selecting the right configuration for the cluster, such as the number of machines (processors) and the resources available on each machine (RAM, I/O devices, network channel, HDD or SSD) [4]. Apache Spark applications are executed in many steps, where each step includes multiple tasks running parallelly on multiple worker nodes. Resilient Distributed Dataset (RDD) [4, 5] provides interfaces for data transformations and parallelization. These RDDs are distributed among cluster nodes. There are two types of operations on RDDs in Spark: transformations, which convert an RDD to another RDD, and actions, which operate on an RDD for producing a final result. This allows creating a job DAG (Directed Acyclic Graph) of transformations before calling an action, and therefore, optimization can be carried out on this DAG prior to execution [2, 6, 8]. For each action triggered inside of a Spark application, the DAG scheduler develops an execution plan to complete it. The conception of this execution plan supposes assembling as many transformations with narrow dependencies as possible into stages [2, 5, 6]. So, The Spark DAG schedules each stage for execution [2, 8]. The RDD interface also provides the possibility of caching data in memory in a way allowing them to be read at next iterations of the job without I/O latency [7, 8].

Spark SQL [8, 9] is a module that is built on top of the Spark core engine in order to process structured or semi-structured data. Spark SQL introduces a novel extensible optimizer called Catalyst [9]. Catalyst makes it easy to add data sources, optimization rules, and data types. There is a possibility of repartitioning data in RDDs. It is possible to do with the help of a specific transformation, such as groupByKey and sortByKey, which leads to a new RDD, where data are differently partitioned across machines. Such a redistribution of data is called data shuffling. An API Data-Frame has been created to perform relational operations on data such as select, filter, and join [2, 9]. Data Frame in Spark is a distributed data collection organized in the form of named columns. Conceptually, this is equivalent to a table in a relational database or data frame in R/Python, but with more powerful optimization. Data frames can be created from different sources – files with structured data, Hive tables, external databases or existing RDD [2, 9]. Data Frames in Spark automatically optimize the execution of queries with the help of the query optimizer Catalyst. At the beginning of calculations in Data Frame Catalyst compiles operations that have been used for building Data Frame in a physical execution plan and then generates byte-code JVM for the plans that are more optimized than a handwritten code [8, 9]. Unlike RDDs, Data Frames normally tracks their schema and support different relational operations, which leads to a more optimized execution. Data Frames can be constructed from tables in the available Big Data infrastructure or from existing RDDs [9]. Data Frames can be managed with direct SQL queries and also with the help of the DataFrame DSL (domain-specific language), where various relational operators and transformers can be used, such as Where and GroupBy. Any Data Frame can also be considered as RDD of Row objects, which allow users to call

RDD of Row objects, which allow users to call procedural Spark APIs such as map [9, 10].

Big Data Benchmarking Requirements include the capability of processing data of different types, e.g., unstructured, semi-structured, structured data, and different sources [9, 11, 13]. Big data workloads selected in the benchmark suite should correspond to the diversity of application scenarios. Some researchers have used 3 benchmark applications: sort-by-key, shuffling and k-means (e.g., part of the HiBench benchmark [11-13]). The above applications have been chosen because they can be viewed as representatives of a variety of applications under the condition that they cover both cpu- and shuffling-intensive cases (Sort-by-key is both computation- and shuffling-intensive) [2, 12, 13]. A great number of transformations require data shuffling across the cluster, which includes Join, ReduceByKey, GroupByKey, Cogroup. In this paper such an approach serves as a basis for the development of a test data generator.

Profiling Spark applications. Many researches divide all methods for tuning Apache Spark into several classes. A brief description of these is as follows:

- using pairs of parameters on benchmarking applications and the application of a graph algorithm to build complex candidate configurations [2, 6];
- using arbitrary combinations of the parameters [2, 3, 8];
- tuning parameters with the help of Machine Learning methods [6-8].

The work [2] considers Spark's shuffling optimization using two approaches. The first one is optimization via columnar compression, which unfortunately does not lead to any significant improvement of performance. The second one applies file consolidation during shuffling. The shuffle phase is all-to-all communication instrument and it can potentially introduce network, disk, memory and CPU scheduling overheads. A lot of transformations require data shuffling across the cluster, including Join, ReduceByKey, GroupByKey, Cogroup. For optimizing shuffle performance the following approaches are used: 1) Emulating Spark behavior by merging intermediate files; 2) Creating large shuffle files [14-16].

The main operations associated with shuffling are represented in Table 2 [15].

Table 2. Shuffle operations

Configuration	Description	Tuning advice
Spark.shufle.file.buffer	This parameter is used to set the buffer buffer size of the bufferedOutputStream of the shuffle write task.	If the available memory resources are sufficient, it can increase the size of this parameter (such as 64k), so as to reduce the number of times the disc file overflows during the shuffle write process, which can reduce the number of disc IO times and improve performance

Spark.reducer.maxSizeInFlight	This parameter is used to set the buffer size of the shuffle read task, and this buffer decides how much data can be drawn at a time.	If the available memory resources are sufficient, we can increase the size of the parameters (such as 96 MB), thereby reducing the number of times the data is pulled, which can reduce the number of network transmissions and improve performance
Spark.shuffle.io.maxRetries	Shuffle read task from the shuffle write task where the node is pulling their own data, if the network due	For those jobs that contain a very time-consuming shuffle operation, it is recommended to increase the maximum number of retries (for example, 60 times) to avoid data failure due to factors such as the full gc of the JVM or network instability.
Spark.shuffle.io.retryWait	This parameter represent the retry interval for each retry of the data).	It is recommended to increase the interval length (e.g. 60s) to increase the stability of the shuffle operation
Spark.shuffle.memoryFraction	This parameter represent the memory size of the Executor memory, which is assigned to the shuffle read task for the aggregation operation.	This parameter is explained in resource parameter tuning. If the memory is sufficient, and rarely use the persistence operation, it is recommended to increase this ratio, to shuffle read the aggregation operation of more memory, in order to avoid the lack of memory caused by the frequent process of reading writing disk.
Spark.shuffle.sort.bypassMergeThreshold	When ShuffleManager is SortShuffleManager, if the number of shuffle read tasks is less than this threshold (default is 200), shuffle write process will not be sorted, but directly in accordance with the way to optimize the HashShuffleManager to write data.	When you use SortShuffleManager, if we do not need to sort operation, it is recommended that this parameter will be larger, greater than the number of shuffle read task.

The contributions of this paper are as follows:

- a technique for generating test files of different sizes with properties defining requirements to test data processing has been developed;

- a comparative performance analysis of using Resilient Distributed Dataset and Data Frame in the case of standard Spark settings has been carried out; the influence of input files of different sizes has been investigated;
- the analysis of the Data Frame technology performance under simultaneously changing settings (tuning) related to parallelism levels and shuffle partitions (Data Frames) for input files of different sizes has been conducted.

2 Problem statement

The problem considered in this paper is the optimization of distributed computations, namely, experimental determining potentially most important tuning parameters for Apache Spark for minimizing the application execution time in the Apache Spark cluster based on using new Data Frame technologies for working with data as com-pared with the technology Resilient Distributed Dataset; obtaining quantitative esti-mates for the application of the API Data Frame technology for different settings (tuning) of the Spark parameters and experimental determining their values providing minimum execution time.

3 Experiments Environment setup and Data Set generation

When conducting experiments, the following software and hardware for the work of the Apache Spark cluster have been used:

4 work stations (1 master, 3 workers) with the following characteristics: CPU: Intel Core i5-7500 CPU @ 3.40 GHz; number of cores: 4 (without HyperThreading);

L1 cache: 256 KB; L2 cache: 1 MB; L3 cache: 6 MB; RAM: 8 GB DDR4 @ 2400 MHz; HDD: 500 GB (1 TB); interface: SATA; computer network – Ethernet; network adapter: Realtek RTL8411@1 GBit/s; OS: Ubuntu Linux 16.0 LTS. For distributed computations on the cluster the following versions of Apache Spark and Apache Hadoop and Spark deployment modes have been used: Master – Spark 3.0.0 (Preview 2), Name Node – Hadoop (HDFS) – 3.2.1, YARN (Fair), client mode; workers – Spark 3.0.0 (Preview 2), Data Node – Hadoop (HDFS) 3.2.1, YARN (Fair), client mode. For testing a dataset containing information on movies from the site https://datasets.imdbws.com has been used (a description of the dataset is presented on the resource https://www.imdb.com/interfaces). The dataset consists of several files linked with a key and contains columns with different data types, such as string, numerical, Boolean and date. The following dataset files have been used for testing: name.basics; title.basics; title.akas; title.episode; title.principals; title.ratings.

The size of the created basic test file is about 4 GB. In order that we can obtain results that can be compared and analyzed, datasets of higher volumes have been generated (Big Data) based on the main dataset. The following method has been used for generating these datasets: the original files have been copied and linked with each other (merged) and the key fields have been changed so that records are considered unique after the application of the aggregation, sorting and joining by key operations. Thus, test data in the range from 8 GB to 16 GB have been obtained. The operations

that have been used for a load imitation have been identical for RDD and Data Frames, namely, filter, join, groupBy, sortBy, and the aggregation functions count and sum. For each file in the dataset the operations of filtering have been ap-plied, then the operations of sorting and aggregations, and after that, the linking by key with another file has been carried out. Thus, the operations that require both computer intensity (processors) and those where input-output speed is a key factor have been applied. The testing has been carried out in two modes: a) using RDD and b) using Data Frames.

For the file name.basics the following filters have been applied: deathDate != null and primaryProfession != miscellaneous. Their implementation though the Data-Frames API is as follows:

```
names_df.filter(f.col('deathYear').isNotNull())
.filter(f.array_contains(f.split(f.col('primaryProfession
'), ','), 'miscellaneous'))
```

and for RDD API:

```
names_rdd.filter(lambda r: r['deathYear'] is not
None).filter(lambda r: False if r['primaryProfession'] is
None else 'miscellaneous' not in r['primaryProfession']).
```

For the file title.akas the following filters have been applied: isOriginalTitle = ‘1’ and region = ‘US’. Their implementation through the DataFrames API is the following:

```
akas_df.filter(f.col('isOriginalTitle') ==
'1').filter(f.col('region') == 'US')
```

and for the RDD API:

```
akas_rdd.filter(lambda r: r['isOriginalTitle'] ==
'1').filter(lambda r: r['region'] == 'US').
```

For the file title.episode the following filter has been applied: episodeNumber > 10. Its implementation through the DataFrames API is as follows:

```
episodes_df.filter(f.col('episodeNumber').cast('int') >
10))
```

and for the RDD API:

```
episodes_rdd.filter(lambda r: False if r['episodeNumber']
is None else int(r['episodeNumber']) > 10).
```

For the file title.principals the following filters have been used: category != ‘self’ and category != ‘cinematographer’ and ordering <= 3. Their implementation thourgh the DataFrames API is as follows:

```
principals_df.filter((f.col('category') != 'self') &
(f.col('category') !=
'cinematographer')).filter(f.col('ordering').cast('int')
<= 3)
```
and for the RDD API:

```
principals_rdd.filter(lambda r: r['category'] != 'self'
and r['category'] != 'cinematographer').filter(lambda r:
int(r['ordering']) <= 3).
```

For the file title.ratings the filters averageRating > 5.0 and numVotes > 10 000 have been applied. Their implementation through the DataFrames API is the following:

```
ratings_df.filter(f.col('averageRating') >
5.0).filter(f.col('numVotes').cast('int') > 10_000)
```

and for the RDD API:

```
ratings_rdd.filter(lambda r: float(r['averageRating']) >
5.0).filter(lambda r: int(r['numVotes']) > 10_000).
```

For the file title.basics the filters ‘Comedy’ in genres and titleType != ‘short’ have been used. Their implementation though the DataFrames API is as follows:

```
tiles_df.filter(f.array_contains(f.split(f.col('genres'),
','), 'Comedy') & (f.col('titleType') != 'short'))
```

and for the RDD API:

```
titles_rdd.filter(lambda r: False if r['genres'] is None
else 'Comedy' in r['genres'].split(',')).filter(lambda r:
r['titleType'] != 'short').
```

After the filtration, the sorting and arrogation operations have been applied. For the file title.principals the operation of sorting by vlues of the column ‘tconst’ has been used, which is implemented through the Data Frames API: principals_df.orderBy('tconst'); in the case of the RDD API: principals_rdd.sortBy(lambda r: r['tconst']). For the file title.basics the operation of grouping by genres has been applied, which is implemented through the Data Frames API: `titles_df.withColumn('genres', f.split(f.col('genres').getItem(0))).groupBy('genres').agg(f.count(f.lit(1)))` и для RDD API: titles_rdd.groupBy(lambda r: `None if r['genres'] is None else r['genres'].split(',')[0]).map(lambda r: (r[0], len(r[1])))`.

The last operation is a superposition of several files by key, which is implemented through the DataFrames API:

```
titles_df.join(akas_df, f.col('tconst') ==
f.col('titleId')).join(ratings_df,
on=['tconst']).join(episodes_df, on=['tconst'])
```

and the RDD API:

```
titles_rdd.join(akas_rdd).join(ratings_rdd).join(episodes
_rdd).
```

All the above operations can be divided into two categories:

CPU intensive (mainly associated with filtering or computing values in the framework of one row);

I/O (operations associated with data exchange between cluster nodes, for example, aggregation or sorting).

The considered operations used for generating test data with different properties are represented in Table 3.

Table 3. Operations for generated data

Operation	CPU intensive	I/O intensive
name.basics	filters deathDate != null and primaryProfession != miscellaneous	superposition with another dataframe by key ‘nconst’
title.akas	filters isOriginalTitle = ‘1’ and region = ‘US’	superposition with another dataframe by key ‘tconst’
title.episode	filters episodeNumber > 10	superposition with another dataframe by key ‘tconst’
title.principals	filters category != ‘self’ and category != ‘cinematographer’ и ordering <= 3	sorting by values of the column ‘tconst’
title.ratings	filters averageRating > 5.0 and numVotes > 10 000	superposition with another dataframe by key ‘tconst’
title.basics	filters ‘Comedy’ in genres and titleType != ‘short’	operations of grouping by genre in the column ‘genres’ superposition with other data frames by the key ‘titleId’

4 RDD vs Data Frame

For conducting experiments and comparative analysis the following setting for Apache Spark parameter tuning have been used (Table 4).

Table 4. Apache Spark settings

Parameters	Description	Default	Rules of thumb	Values
spark.driver.cores	Number of cores to use for the driver process	1	1-4	1
spark.driver.memory	Amount of memory to use for the driver process	1 GB	1-8 GB	1 GB
spark.executor.cores	The number of cores to use on each executor	All the available cores on the worker	1-4	3
spark.executor.memory	Amount of memory to use per executor process	1 GB	1-8 GB	6 GB
spark.io.compression.codec	The codec used to compress internal data	lz4	lz4/lzf	lz4
spark.reducer.maxSizeInFlight	Maximum size of map outputs to fetch simultaneously from each reduce task	48 MB	24-48 MB	48 MB
spark.shuffle.compress	Whether to compress map output files	true	true/false	true
spark.default.parallelism	Default number of partitions in RDDs returned by transformations	Number of cores on the local machine	9-72	9-72
spark.sql.shuffle.partitions	Number of partitions to use when shuffling data for joins or aggregations	200	50-200	50-200
spark.sql.shuffle.buffer	Size of the in-memory buffer for each shuffle file output stream, in KB unless otherwise specified. These buffers reduce the number of disk seeks and system calls made in creating	32	16-64	64

	intermediate shuffle files			
input data size	Total size of input data files	–	4-16 GB	4-16 GB
hdfs.blocksize	Size of blocks in which files are stored in hdfs	128 MB	64-256 MB	64-256 MB

The conducted experimental research has shown the absence of influence of changing standard values for some tuning Spark parameters, which are shown in Table 4, namely, spark.shuffle.compress, spark.reducer.maxSizeInFlight, spark.driver.memory and spark.io.compression.codec on the execution time. The value of the parameter spark.sql.shuffle.buffer (64 MB) has been obtained based on multiple experimental results for RDD and Data Frames. Therefore the presented methodology has been built on the investigation of the parameters input data size, spark.default.parallelism, spark.sql.shuffle.partitions, and hdfs.blocksize on the task execution time.

The experimental research included two stages: at the first stage a comparison of test results for input files of 4GB – 16GB has been carried out for different Spark settings in the cases RDD and Data Frame; at the second stage the influence of parallelism levels and the parameter spark.sql.shuffle.partitions on the execution time for files with different sizes for different values of the HDFS data block has been investigated.

As a performance metric, the application execution time (*Execution Time*) has been used. It can be calculated according to formula [17]:

$$Execution\ Time = Finish\ Time - Start\ Time\ ,$$

where *Finish Time* is the application completion time; *Start Time* is the application start time.

Stage 1.

The experiments have been carried out for different sizes of the HDFS data block (by default 128 MB). The results obtained for the test task execution time in the cases of RDD and Data Frames are shown in Table 5.

Table 5. Task execution time for RDD and Data Frames for different values of input files sizes and levels of parallelism

Type	Input file size, GB	Level of parallelism	Execution time
RDD	4	9	8 min. 35 sec.
	4	36	7 min. 35 sec.
	4	72	6 min. 59 sec.
	8	9	38 min. 45 sec.
	8	36	17 min. 28 sec.
	8	72	14 min. 51 sec.
	16	9	49 min. 23 sec.

	16	36	40 min. 45 sec.
	16	72	31 min.28 sec.
Data Frame	4	9	1 min. 37 sec.
	4	36	1 min. 53 sec.
	4	72	1 min. 33 sec.
	8	9	2 min.54 sec.
	8	36	2 min. 58 sec.
	8	72	3 min. 29 sec.
	16	9	5 min. 3 sec.
	16	36	5 min.19 sec.
	16	72	5 min. 13 sec.

The analysis of results shown in Table 5 allows us to come to a conclusion that the performance of computations with the help of Data Frame is significantly higher than with the using of RDD (in the range of 600%-700%), and it changes with changing the level of parallelism as follows: it decreases for RDD in the range of 15%-230% and it increases for Data Frames. The latter result shows the necessity of investigating the influence of other factors (parameters) used in Data Frames on the performance of computations.

Stage 2.

At this stage the combined influence of changing the parameter settings Level of Parallelism and spark.sql.shuffle.partitions on the application execution time has been investigated. A peculiarity of the suggested approach is this. The parameter spark.sql.shuffle.partitions decreases (it is equal to 200 by default) in the range from 50 to 200. The results of the conducted computing experiments are shown in Table 6.

Table 6. Task execution time for Data Frames for different values of input files sizes, levels of parallelism, and spark.sql.shuffle.partitions

Level of Parallelism	spark.sql.shuffle.partitions	Execution Time
Input File size=8 GB, Block size=128 MB		
9	200	2 min.54 sec.
	150	2 min.45 sec.
	100	2 min.52 sec.
	50	2 min.40 sec.
36	200	2 min.58 sec.
	150	2 min.48 sec.
	100	2 min.52 sec.
	50	2 min.43 sec.
72	200	3 min.29 sec.
	150	2 min.48 sec.
	100	2 min.43 sec.
	50	2 min.45 sec.
Input File size=8 GB, Block size=64 MB		

9	200	3 min. 7 sec.
	150	2 min. 49 sec.
	100	2 min. 35 sec.
	50	2 min. 48 sec.
36	200	2 min. 29 sec.
	150	2 min. 28 sec.
	100	2 min. 24 sec.
	50	2 min. 21 sec.
72	200	2 min. 21 sec.
	150	2 min. 21 sec.
	100	2 min. 18 sec.
	50	2 min. 19 sec.
Input File size=4 GB, Block size=128 MB		
9	200	1 min. 37 sec.
	150	1 min. 38 sec.
	100	1 min. 34 sec.
	50	1 min. 35 sec.
36	200	1 min. 53 sec.
	150	1 min. 31 sec.
	100	1 min. 32 sec.
	50	1 min. 28 sec.
72	200	1 min. 33 sec.
	150	1 min. 32 sec.
	100	1 min. 30 sec.
	50	1 min. 30 sec.
Input File size=4 GB, Block size=64 MB		
9	200	1 min. 58 sec.
	150	1 min. 44 sec.
	100	1 min. 41 sec.
	50	1 min. 36 sec.
36	200	1 min. 34 sec.
	150	1 min. 34 sec.
	100	1 min. 31 sec.
	50	1 min. 31 sec.
72	200	1 min. 36 sec.
	150	1 min. 34 sec.
	100	1 min. 36 sec.
	50	1 min. 33sec.
Input File size=16 GB, Block size=128 MB		
9	200	5 min. 3 sec.
	150	5 min. 2 sec.
	100	5 min. 7 sec.
	50	4 min. 59 sec.

36	200	5 min. 19 sec.
	150	5 min. 13 sec.
	100	5 min. 9 sec.
	50	5 min. 11 sec.
72	200	5 min. 13 sec.
	150	5 min. 11 sec.
	100	5 min. 11 sec.
	50	5 min. 11 sec.

The obtained results show that for relatively small test files (4 GB and 8 GB) values of the parameter spark.sql.shuffle.partitions have the maximum influence on the execution time; at that the optimal parallelism level is from 36 to 72 and the minimum execution time is stable when spark.sql.shuffle.partitions equals 50. We should note that for the same files the optimal HDFS data block size is 64 MB or 128 MB, i.e. it practically does not have any influence on the target variable. For a 16 GB file the same settings do not bring substantial results. Besides, the most important result of the conducted research and computational experiments is the fact that in all experiments the default value of spark.sql.shuffle.partitions leads to worse results.

The graphs for the dependence of the test task execution time on parallelism level and the parameter spark.sql.suffle.partitions are shown in Fig. 1, 2 (the following notation is used: IFS – Input File Size, BS – Block Size – block size in HDFS).

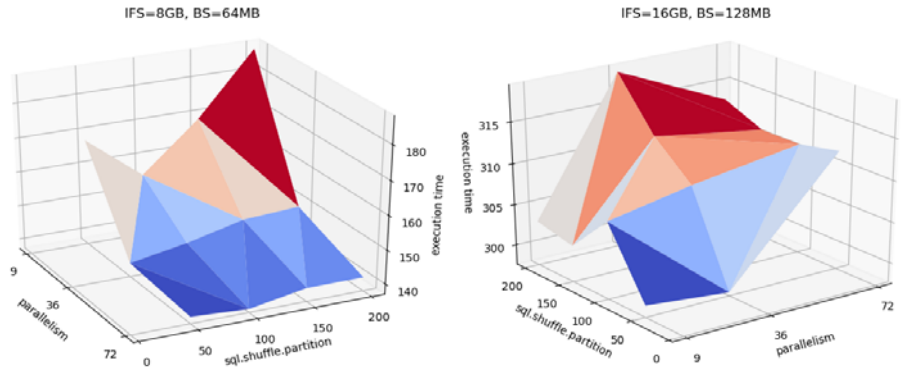

Fig. 1. Graphs for the dependence of the test task execution time on different values of BS for IFS=8 GB.

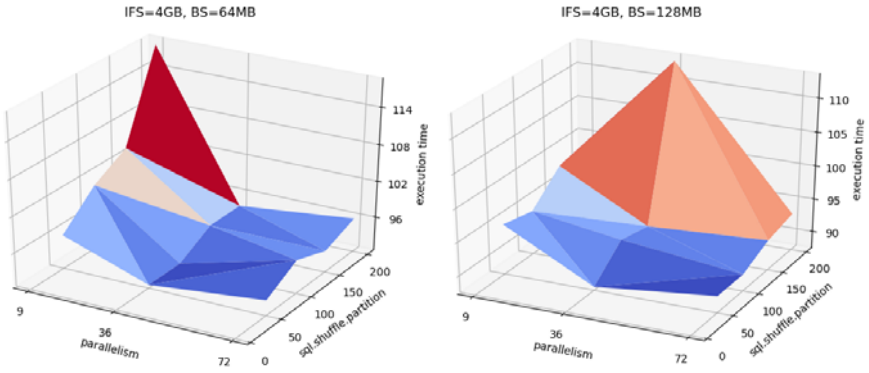

Fig. 2. Graphs for the dependence of the test task execution time on different values of BS for IFS=4 GB.

On graphs 1, 2, results of conducted execution time calculations are visualized. These results should be interpreted as follows. The minimum execution time values are shown in blue and its gradient, the maximum ones are shown in red and its gradient. The results confirm the fact (see Table 6) that in general, for small size files minimum execution time values are determined when the parallelism levels are 36 or 72, and the parameter sql.spark.shuffle.partions equals 50.

5 Conclusions

A new technology for working with data in Apache Spark – API Data Frames – has been investigated. A comparative analysis of performance for RDDs and Data Frames in Apache Spark has been conducted on test data with properties defined by different data volumes. Software for generating test data with pre-defined characteristics has been developed and studied, taking into consideration CPU and I/0 intensity. A comparative analysis of the technologies RDD and Data Frames has been carried out, which has shown a significant advantage of Data Frames and the influence of the parallelism level as the most important factor of increasing performance of computations for RDD for the standard Spark settings. By conducting a series of experiments, it has been obtained that the most important parameter for the Data Frame technology that influences the execution time is spark.sql.shuffle.partitions. Its changes in the range from 50 (experimentally considered to be the minimum value) to 200 (default value) have shown that for input files of small sizes the default value gives the worst results, and increasing the performance of computations is determined by decreasing the values of this parameter.The investigation of the parallelism level influence has allowed determining optimal (in the context of settings for minimizing execution time) values for this parameter - 36 and 72. For big size files decreasing the value spark.sql.shuffle.partitions and increasing the parallelism level wihin the selected ranges does not practically lead to a decrease in execution time. This fact probably means that when an input file size increases, it is necessary to increase the

spark.sql.shuffle.partitions value. At the same time, it is quite difficult to assess the influence of the parallelism level and directions of its change.

The conducted research has demonstrated the topicality of solving Big Data problems in the framework in-memory Apache Spark [18-20], for which the optimization of settings differs from that for small files. Using well-known benchmarkings for assessing distributed computations performance [11, 13, 18, 19] does not always allow determining an optimal set and values of the Spark tuning parameters for RDDs and Data Frames in the package and stream modes. Therefore, there are unsolved problems related to investigating conditions for minimizing execution time for big size files and saving properties of files generated synthetically, according to the considered technique – forming big size files by cloning small files.

In the future we are planning to test the above technologies for data with volumes higher than 50 GB, maybe 100 GB, under the condition of the scalability of the Apache Spark cluster. An important research direction is also associated with investigating the performance of the considered technologies when they are used for methods and tests in maching learning [19-21].

References

1. Wang, K., Khan, M. M. H., Nguyen, N., Gokhale, S.: A Model Driven Approach towards Improving the Performance of Apache Spark Applications. In: 2019 IEEE Interna-tional Symposium on Performance Analysis of Systems and Software (ISPASS), 24-26 March 2019, pp. 233-242 (2019) doi: 10.1109/ISPASS.2019.00036.
2. Gounaris, A., Torres, J.: A Methodology for Spark Parameter Tuning. Big Data Research, vol. 11, pp. 22-32 (2018) https://doi.org/10.1016/j.bdr.2017.05.001.
3. Petridis, P., Gounaris, A., Torres, J.: Spark Parameter Tuning via Trial-and-Error. In: Advances in Big Data: Proceedings of the 2nd INNS Conference on Big Data. INNS 2016. Advances in Intelligent Systems and Computing, vol. 529, pp.226-237 (2016) https://doi.org/10.1007/978-3-319-47898-2_24.
4. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud), pp. 1-7 (2010) https://www.icsi.berkeley.edu/icsi/node/5074.
5. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, Franklin, M. J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, pp. 15-28 (2012) doi/10.5555/2228298.2228301.
6. Wang, G., Xu, J., He, B.: A Novel Method for Tuning Configuration Parameters of Spark Based on Machine Learning. In: 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Sydney, NSW, pp. 586-593 (2016) doi: 10.1109/HPCC-SmartCity-DSS.2016.0088.
7. Choi, I. S., Yang, W., Kee, Y.: Early experience with optimizing I/O performance using high-performance SSDs for in-memory cluster computing. In: IEEE International Confer-

ence on Big Data (Big Data), pp. 1073-1083 (2015) doi: 10.1109/BigData.2015.7363861.

8. Mustafa, S., Elghandour, I., Ismail, M. A.: A Machine Learning Approach for Predicting Execution Time of Spark Jobs. Alexandria Engineering Journal, vol. 57, issue 4, pp. 3767–3778 (2018) https://doi.org/10.1016/j.aej.2018.03.006.
9. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, Michael J., Ghodsi, A., Zaharia, M.:Spark SQL: Relational Data Processing in Spark. In: SIGMOD '15: Proceedings of the 2015 ACM SIGMOD International Conference on Man-agement of Data, pp. 1383–1394 (2015) https://doi.org/10.1145/2723372.2742797.
10. Zaharia, M., Xin, R., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M., Ghodsi. A., Gonzalez, J., Shenker, S., Stoica, I.: Apache Spark: a unified engine for big data processing. Communications of the ACM, vol. 59, no. 11, pp. 56-65 (2016) https://doi.org/10.1145/2934664.
11. Han, R., John, L. K., Zhan, J.: Benchmarking Big Data Systems: A Review. IEEE Transactions on Services Computing, vol. 11, issue 3, pp. 580-597 (2018) doi: 10.1109/TSC.2017.2730882.
12. Clemente-Castello, F., Nicolae, B., Mayo, R., Fernandez, J.: Performance Model of MapReduce Iterative Applications for Hybrid Cloud Bursting. IEEE Transactions on Parallel & Distributed Systems, vol. 29, issue 8, pp. 1794-1807 (2018) doi: 10.1109/TPDS.2018.2802932.
13. Huang, S., Huang, J., Dai, J., Xie, T., Huang, B.: The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In: 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010), Long Beach, CA, pp. 41-51 (2010) doi: 10.1109/ICDEW.2010.5452747.
14. Davidson, A., Or, A.: Optimizing shuffle performance in spark. University of California, Berkeley Department of Electriacal Engenering and Computer Science (2013) https://www.semanticscholar.org/paper/Optimizing-Shuffle-Performance-in-Spark-Davidson/d746505bad055c357fa50d394d15eb380a3f1ad3.
15. Nirali, R., Shyam, D.: Shuffle Performance in Apache Spark. International Journal of Engineering Research & Technology (IJERT), vol. 4, issue 02, pp.177-180 (2015).
16. Vandana, Velamuri, M., Narendra, K. P.: Shuffle phase optimization in spark.In: 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Udupi, pp. 1028-1034 (2017) doi: 10.1109/ICACCI.2017.8125977.
17. Web UI, available at: https://spark.apache.org/docs/3.0.0-preview/web-ui.html#jobs-tabN, last accessed 2020/04/20.
18. Li, M., Tan, J., Wang, Y. et al.: SPARKBENCH: a spark benchmarking suite characterizing large-scale in-memory data analytics. Cluster Computing, vol. 20, no. 3, pp. 2575–2589 (2017) https://doi.org/10.1007/s10586-016-0723-1.
19. Rodrigues, J., Vasconcelos, G.: Big Data Machine Learning Benchmark on Spark. IEEE Dataport (2019) http://dx.doi.org/10.21227/t8bg-yc46.
20. Shi, S., Wang, Q., Xu, P., Chu, X.: Benchmarking State-of-the-Art Deep Learning Software Tools. In: 7th International Conference on Cloud Computing and Big Data (CCBD), Macau, pp. 99-104 (2016).
21. Liu, Y., Zhang, H., Zeng, L., Wu, W., Zhang, C.: MLbench: benchmarking machine learning services against human experts. Proceedings of the VLDB Endowment, vol. 11, no. 10, pp. 1220–1232 (2018) doi:https://doi.org/10.14778/3231751.3231770.